# Nomad: A Scalable Operating System for Clusters of Uni and Multiprocessors *

Eduardo Souza de Albuquerque Pinheiro† and Ricardo Bianchini‡

COPPE Systems Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil 21945-970

{edpin,ricardo}@cos.ufrj.br

**Abstract**

The recent improvements in workstation and interconnection network performance have popularized the clusters of off-the-shelf workstations. However, the usefulness of these clusters is yet to be fully exploited, mostly due to the inadequate management of cluster resources implemented by current distributed operating systems. In order to eliminate this problem and approach the computational power of large clusters of workstations, in this MSc thesis we propose Nomad, an efficient operating system for clusters of uni and/or multiprocessors. Nomad includes several important characteristics for modern cluster-oriented operating systems: scalability, efficient resource management across the cluster, efficient scheduling of parallel and distributed applications, distributed I/O, fault detection and recovery, protection, and backward compatibility. Some of the mechanisms used by Nomad, such as process checkpointing and migration, can be found in previously proposed systems. However, our system stands out for its strategy for disseminating information across the cluster and its efficient management of *all* cluster resources. In addition, Nomad is highly scalable as it uses neither centralized control nor extra messages to implement its functionality, taking advantage of the I/O traffic associated with its distributed file system. Our evaluation of the load balancing aspect of Nomad shows that the pattern of file accesses in our distributed file system allows for efficient and scalable load balancing. Our main conclusion is that Nomad will be an interesting and useful platform for future research on operating systems for clusters of workstations.

## 1 Introduction

The recent improvements in workstation and network performance have popularized the clusters of off-the-shelf workstations as a platform for both high-performance and interactive applications. Clusters of workstations currently have the potential to approach the performance of supercomputers and the ease of use of mainframes with much cheaper hardware. However, the main obstacles yet to be surpassed by these clusters as general-purpose computing platforms are two deficiencies of most current distributed operating systems: their inadequate management of the available resources and their inability to present an integrated view that can simplify utilization, programming, and management of the cluster as a whole.

Current systems invariably focus on the management of some resources in detriment of others and are usually not scalable to large cluster configurations. For instance, the systems based on load balancing (e.g. Sprite [6] and V [4]) consider the number of processes per workstation, but disregard the memory and I/O behavior. Another example of poor resource management is the scheduling of parallel and distributed applications: efficient scheduling strategies, such as co-scheduling [11], are simply not implemented in most systems (e.g. Mosix [2]). Even the systems that apply sophisticated load balancing

*A similar version of this paper was published in the Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing, December 1999.

†Eduardo Pinheiro is now a PhD student at the University of Rochester.

‡Ricardo Bianchini is currently on leave from the Federal University of Rio de Janeiro.

techniques (e.g. Mosix) or intelligently schedule applications (e.g. GLUnix [8]) use centralized servers and/or extraneous messages for these purposes. In fact, the number of extra messages involved in these systems is proportional to the number of workstations in the cluster. Other examples of inadequate management of resources abound.

Current systems can also be improved to simplify cluster use, programming, and management. Ideally, the system should provide a cluster-wide single-system image, such that remote resources can be handled transparently as if they were local. The problem is that most distributed operating systems do not attempt to hide the fact that the cluster hardware is really composed of multiple independent workstations and resources. For instance, in most systems the user must explicitly connect to another workstation in order to have access to its resources (e.g. Solaris [13]).

In the thesis (MSc at the Federal University of Rio de Janeiro) we introduce Nomad, a distributed operating system for clusters of uni and/or multiprocessors that eliminates these problems. The main goal of Nomad is to efficiently support (high-performance or interactive) parallel, distributed, and sequential applications. Besides the characteristics required of all operating systems, such as protection between processes and backward compatibility with old binaries, Nomad has several important characteristics: it simplifies the use, programming, and management of the cluster; it manages *all* cluster resources (CPUs, memories, and I/O devices); it is highly scalable and efficient; and it provides tolerance and recovery to workstation failures.

The mechanisms used to implement these characteristics include unique cluster-wide process identifications, process checkpointing and migration, co-scheduling of concurrent applications, and a distributed file system. Some of these mechanisms, such as process checkpointing and migration, can be found in other systems. However, Nomad is unique in the particular set of characteristics it includes, in its strategy for disseminating information across the cluster, and in that it manages all cluster resources, while using neither extra messages nor centralized servers to implement its functionality. Nomad avoids sending extra messages by relying on the communication intrinsic to its distributed file system (which is required for high disk I/O throughput and fault tolerance anyway). For instance, in order to implement process migration, load information is piggybacked on file access messages.

This paper presents a short introduction to Nomad, its main characteristics, and performance. A complete evaluation of all Nomad's policies and mechanisms is part of the thesis, but is not presented in this paper due to space limitations. In terms of performance, here we focus solely on the load balancing aspect of Nomad. A preliminary evaluation of this aspect in the context of a prototype implementation of Nomad shows that the pattern of file accesses produced by Nomad's distributed file system and real workloads can effectively be used as a mechanism for distributing load information across the cluster. In addition, our results show that Nomad can almost eliminate the periods of excessive demand for resources by intelligently migrating processes. Based on these results and on our experience with the other mechanisms implemented in our system, we believe that Nomad will most likely be an efficient, useful, and user-friendly operating system for clusters of uni and multiprocessors.

# 2 Nomad

## 2.1 Functionality

**Single-system image.** Nomad simplifies the use, programming, and management of the cluster by providing a single-system image of it. The user can utilize the system as if it were a single very powerful workstation. This integrated view is based on cluster-wide unique process identifications and on making all aspects of process management (signal delivery, for instance) independent of where processes are actually running.

**Efficient and complete resource management.** Nomad efficiently supports high-performance and interactive applications with its efficient resource management and scheduling. The distribution of resource demands is based on the intelligent initial assignment of processes to processors and on dynamic pro-

cess migration. When launching a new application, Nomad chooses a lightly loaded workstation to host the new process(es). But with time, if a workstation becomes overloaded (i.e. one of its resources is exhausted), Nomad chooses the application consuming the most of the exhausted resource to be migrated to another workstation. The migration itself is initiated by the overloaded workstation, which sends the chosen application to a destination workstation that is lightly loaded with respect to the resource. In order to reduce the number of times multiple overloaded workstations migrate processes to the same destination, each source picks a destination randomly out of the set of workstations that are lightly loaded with respect to the exhausted resource.

The whole image of the application is migrated to the destination workstation and future system calls are executed at the destination workstation. When making its assignment and migration decisions, Nomad considers all aspects of a workstation's load: demand for CPU, memory, disk I/O, and network I/O.

**Efficient process scheduling.** Process scheduling in Nomad targets high performance in many ways. Sequential applications running on multiprocessors are scheduled considering the affinity of each process for the processor on which it ran last. Concurrent applications are co-scheduled [11] or implicitly co-scheduled [1]. In co-scheduling all processes belonging to a parallel application (defined as a concurrent application running on a multiprocessor) are scheduled simultaneously. Implicit co-scheduling is an approximation of co-scheduling for processes of a distributed application (defined as a concurrent application with processes scattered across many workstations). In contrast with other implementations of co-scheduling (e.g. [3]), in implicit co-scheduling all scheduling decisions are made locally by each workstation, without the need for coordination messages or a centralized controller.

**Scalability.** The scalability of the system is guaranteed since it does not involve centralized servers or extra messages in its management of resources, scheduling of distributed applications, and fault tolerance and recovery. In addition, Nomad includes a distributed and redundant file system that provides high-performance I/O by striping files at the block level across the different disks in the cluster. In essence, our distributed file system can be seen as a software implementation of RAID [5], where each block is assigned to a randomly chosen disk, like in the RAMA file system [10]. This assignment of blocks leads to high disk I/O throughput, while avoiding communication bottlenecks [10]. The redundancy in the file system allows for fault tolerance. (Note that files that require neither high throughput nor fault tolerance can be stored locally, bypassing the distributed file system.)

It is important to observe that the distributed file system forces a workstation that needs to access a file to communicate with a potentially large number of other workstations in the cluster, as opposed to a single workstation as in NFS-style file systems. Based on this observation, we realized that Nomad could avoid extra messages in implementing resource management and fault tolerance, by appending the information that must be disseminated through the cluster to the file access messages. Essentially, Nomad avoids sending extra messages by extending each file system message with a few extra bytes.

**Efficient dissemination of load information.** An example of this piggybacking of messages occurs when Nomad uses file access messages to disseminate the load information necessary to perform process migration. The load information of each workstation is sent on its file access messages. Thus, a file access request informs the replier workstation of the requester's load information, while the access reply informs the requester of the replier's load information. (As a fall-back strategy, a workstation running Nomad multicasts its load information to a few other workstations, if it has gone too long – 30 minutes, say – without communicating with any other node.)

Note however that the motivation for using the file access pattern to guide the dissemination of load information is not restricted to the desire to avoid extra messages; another reason is that using this pattern seems like a natural strategy to support load balancing. More specifically, the file access pattern has two relevant properties as a mechanism for distributing load information: (a) the communication between workstations occurs in bi-directional (request/reply) form, as necessary for migration; and (b) idle workstations (which can be numerous) do not generate messages. Under a striped file system such as Nomad's, the file access pattern has the additional property that a significant number of workstations will likely receive file access messages from each non-idle workstation. These three characteristics and the absence of

extra messages make process migration in Nomad potentially more efficient than in other systems (e.g. Mosix, which is in fact very efficient in terms of migration).

**Fault tolerance.** Nomad is capable of detecting the failure of one workstation and exclude it from the cluster until the failure disappears or is repaired. Failure detection is associated with the file access communication involved in Nomad's distributed file system. If a replier fails to reply to a file access request after a timeout and retransmissions period, the replier is considered faulty and the access is diverted to a redundant disk. Any future messages by the requester will now inform other workstations about the failure. A workstation that is informed about a failure must then destroy any local processes belonging to distributed applications affected by the failure. When a faulty workstation resumes normal operation, Nomad tries to recover by adding the workstation back to the cluster, reconstructing the disk according to the redundancy information, and restoring the processes that were running on the workstation prior to the failure. The processes belonging to distributed applications are the only ones that are not restored automatically.

## 2.2 Architecture

The main goal of the architecture of Nomad is to make it as portable and fault tolerant as possible, but without compromising our desired functionality. Thus, we decided to divide the Nomad architecture into two components: a modified version of a Unix operating system and a layer of user-level software (middleware). As modifying the base operating system kernel significantly would reduce the portability of Nomad, we decided to keep kernel modifications to a minimum. Basically, all we do to the base kernel is enlarge it with code to implement process checkpointing and code implementing a few new system calls. The checkpointing code can checkpoint whole applications, regardless of whether they have open files, pipes, semaphores, shared memory segments, or access shared libraries.

Since the base operating system interface is a subset of the Nomad interface, the users and applications can still utilize the base kernel directly, bypassing Nomad altogether, which is useful for backward compatibility. In addition, each copy of the modified base kernel remains independent of copies running on other workstations, thus promoting fault tolerance and easier cluster reconfiguration.

The user-level software is composed by a daemon (called the Nomad daemon), standard I/O redirection daemons, and a set of tools to allow the users and applications to interact with the daemon. Each of the workstations in the cluster runs a copy of the (modified) base operating system and one Nomad daemon. The daemon runs in the background with super-user privileges. The daemon performs several important tasks: (1) it maintains the state of the applications running on top of Nomad; (2) it collects statistics about the use of local resources; (3) it implements the load balancing policies and mechanisms; (4) it implements the process scheduling policies; (5) it interacts with the user and remote daemons for process launching and migration, and distributed signal delivery; and (6) it launches the standard I/O redirection daemon for each application that is launched or migrated away from a user's workstation. Note that, even though the Nomad daemon does not interfere with processes launched directly on top of the base operating system, it does take their resource usage into consideration.

The standard I/O redirection daemons redirect the standard input, output, and error streams to the terminal where the user started the corresponding application. Both the source and destination of the application get a redirection daemon.

The last component of the middleware is the tools used by users and applications to interact with the Nomad daemon. The two main tools are the $netspawn$ and $netkill$ commands. Netspawn is used to launch applications on top of Nomad, while netkill is used to send signals to processes launched by Nomad. By default, netspawn interacts with the local daemon, which intelligently selects a workstation for the application to run on. The main argument to netspawn is the application's name, but the user can also specify that the application should not be migrated or, for a distributed application, specify the number and addresses of workstations to be used. The processes launched with netspawn have cluster-wide unique identifications that are independent of where they are running.

| Machine | # Procs | # Disks | Total/Available Mem (MB) |
|---|---|---|---|
| atto | 1 | 1 | 64/58 |
| brain | 4 | 1 | 2048/1998 |
| gin | 1 | 1 | 64/58 |
| kilo | 1 | 1 | 64/58 |
| ripple | 1 | 1 | 64/58 |
| rum | 1 | 1 | 64/58 |
| rye | 1 | 1 | 64/58 |
| scotch | 1 | 1 | 64/58 |
| vodka | 1 | 1 | 64/58 |

Table 1: Configuration of workstations in the cluster.

Netkill interacts with the local Nomad daemon requesting that a signal be delivered to a certain unique process identification. The daemon is responsible for checking its internal tables and determining where the process is running.

### 2.3 Implementation Status

We now have a prototype implementation of Nomad up and running. The prototype currently supports the x86/Pentium family of microprocessors using Linux as the base operating system and the Alpha family of microprocessors using Digital Unix as the base operating system. Our development platform is an 8-workstation cluster of 4-processor SMPs.

Unfortunately, in the current prototype the dissemination of load information is still implemented with periodic multicast messages, instead of being piggybacked with file access messages. Due to this limitation, here we present a simulation of the load balancing behavior of a complete implementation of Nomad.

## 3 Evaluating Load Balancing

As mentioned in the previous section, Nomad uses the file access patterns to guide the dissemination of load information among workstations in the cluster. However, in order for this strategy to be useful, we must confirm two of our previous claims that: (a) clusters may suffer from uneven demands for resources; and (b) under the Nomad striped file system, a significant number of workstations will likely receive load information from each non-idle workstation. In other words, we must confirm that there is an imbalance problem to be solved and that file access patterns would spread load information widely enough throughout the cluster. Furthermore, we would like to know whether this load dissemination strategy coupled with migration (from overloaded to lightly loaded or idle workstations) can improve the performance of real workloads. These are the topics of this section.

In order to address these issues, we studied the workload of a production, academic environment. More specifically, we logged the user-generated file accesses and the (CPU, memory, and disk I/O) load information of 9 workstations from the main computing laboratory at the University of Rochester for 7 days (from 5/5/99 until 5/11/99). The list of workstations and their configurations is shown in table 1. All workstations have processors from the Ultra family and were running Solaris version 5.5.1.

During the tracing period, users were allowed to freely use these workstations. The typical workload was a mix of different types of applications, such as text editors, web browsers, compilers, simulators, and scientific programs, which matches our target environment.

Let us start by determining what is the demand on the various resources. Figures 1, 2, and 3 show the demand for CPU, memory, and disk I/O resources, respectively, for representative workstations during the tracing period.

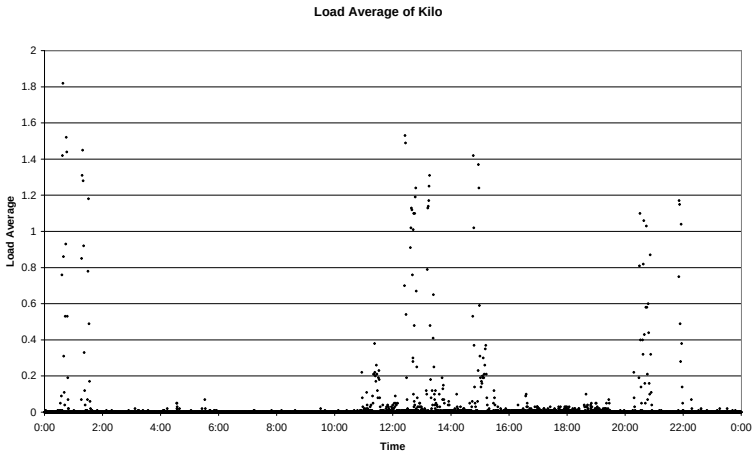


Figure 1: CPU requirements on Kilo.

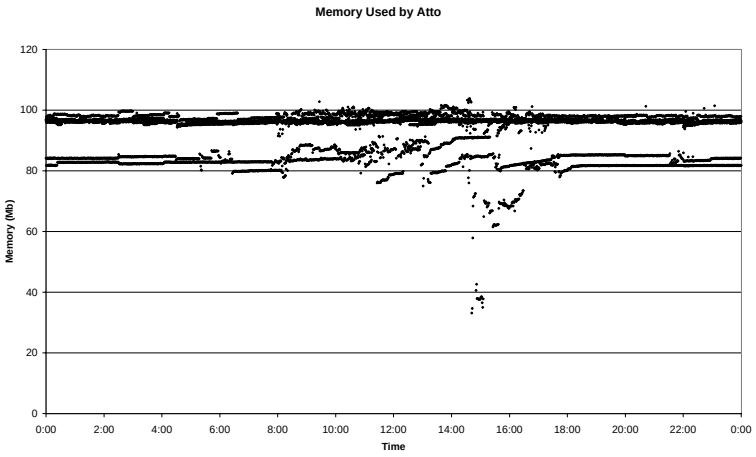


Figure 2: Memory requirements on Atto.

Figure 1 shows the load average for workstation Kilo as a function of time. Each minute plotted has 7 load averages because the different 24-hour periods were overlaid on top of each other. This figure indicates that Kilo experienced load averages that are greater than the ideal load of 1.0 several times during the tracing period. However, even in these periods of high CPU demand, the stress put on the system was not substantial. The same type of CPU behavior was observed of other workstations in the system, suggesting that a high demand for CPU resources was not a major problem during the tracing period.

Figure 2 shows the memory requirements (i.e. the total amount of virtual memory actually used by applications) of workstation Atto as a function of time. Again, the 24-hour periods are overlaid in the figure. The figure indicates that a high demand for memory *may* be a serious problem for Atto, given that its amount of available physical memory is only 58 MBytes. A few other workstations in the system exhibited the same type of memory behavior as this workstation, indicating that the memory requirements was a potential performance problem for the cluster during the tracing period.

Figure 3 shows the disk I/O requirements (disk interrupts/minute) of workstation Ripple as a function of time. Again, the 24-hour periods are overlaid in the figure. The figure indicates that Ripple experienced low I/O demands throughout the tracing period. The other workstations exhibited the same type of behavior. Thus, a high demand for I/O resources was not a problem during the tracing period.

These figures demonstrate that the only resource that experienced significant demands was main memory; other resources were rarely intensively used. Excessive demands for memory may lead to frequent

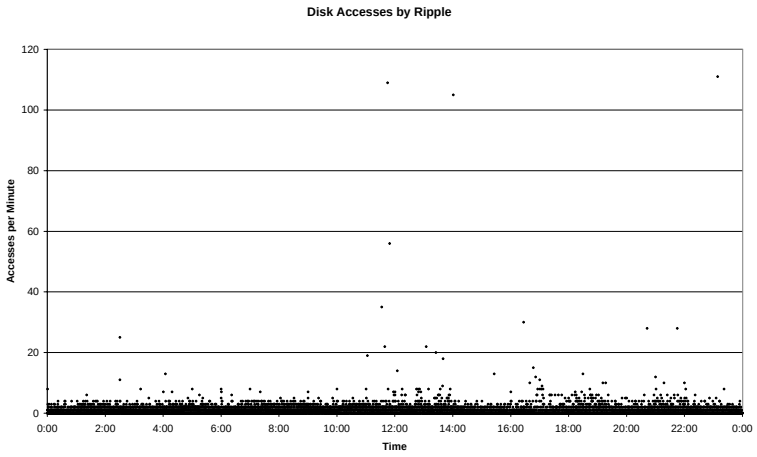


Figure 3: Disk I/O requirements on Ripple.

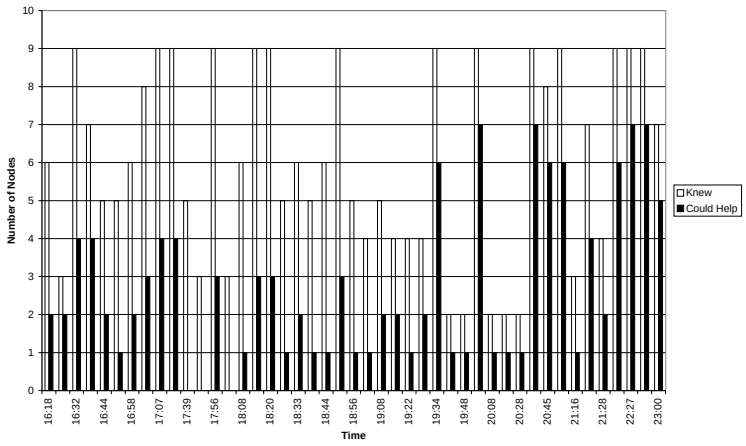


Figure 4: Number of workstations that learned about and could alleviate the excessive demand on Gin.

page replacements and possibly accesses to disk (page swap-outs). Thus, a system that could more evenly balance the memory requirements of the cluster should be useful. To confirm this claim and investigate whether the Nomad strategy for disseminating load information would allow for good load balancing, we simulated the behavior of the Rochester workload, assuming the Nomad striped file system.

With the information we collected, we were able to simulate the behavior of our striped file system. Each file access in the traces was fed to a striping function and a new target workstation was computed, based on the file name and access position. A modified log was then generated with the new target workstations of all file accesses. This new log effectively describes the pattern of file accesses and load information exchanges had our file system indeed been used.

Figure 4 shows the number of workstations that were aware of (white bar) and could alleviate (dark bar) the "excessive" demands on workstation Gin. We decide whether demands for CPU, memory, and disk I/O resources are excessive based on three thresholds: 1.05 load average, less than 3 MBytes of available physical memory (point at which most Unix implementations start running their page replacement algorithms for 64-MByte memories), and more than 2100 disk interrupts per minute (70% disk utilization approximately), respectively. Workstations that could alleviate Gin's problems had plenty of the exhausted resource available. Each pair of bars corresponds to one 5-minute period of excessive demand during the 7-day tracing period.

| Machine | $< 1$ min | $1 - 5$ mins | $5 - 20$ mins | $> 20$ mins |
|---|---|---|---|---|
| Atto | 1 | 7 | 10 | 25 |
| Brain | 0 | 0 | 0 | 0 |
| Gin | 12 | 18 | 8 | 12 |
| Kilo | 2 | 10 | 2 | 1 |
| Ripple | 2 | 3 | 3 | 12 |
| Rum | 4 | 5 | 7 | 12 |
| Rye | 8 | 27 | 10 | 12 |
| Scotch | 6 | 8 | 12 | 5 |
| Vodka | 12 | 22 | 11 | 16 |
| Total | 47 | 100 | 63 | 95 |

Table 2: Duration of the periods of overdemand.

The figure shows that more than 92% of the time at least one workstation not only knew about the problem with Gin, but also could have taken some of its load. Other workstations exhibit similar results to Gin. In the worst case, 87% of the time there was at least one workstation ready to help. These results clearly show that Nomad's strategy for disseminating load information should allow for good load balancing, since load information is spread widely enough for load to be balanced more evenly.

However, for migration to be useful we must verify that periods of excessive demand persist long enough to offset the migration overhead. Table 2 lists the number of these periods according to their different durations: less than one minute, from one minute to five minutes, from five minutes to twenty minutes, and more than twenty minutes. These results show that the vast majority (85%) of all overdemand periods last for more than one minute. Overdemand durations of more than one minute are long enough to offset the overhead of migration, even when processes have very large checkpoints [12]. Furthermore, note that in terms of time, the overdemand periods of one minute or less account for a negligible percentage of the total overdemand time.

Based on these positive results, we simulated the behavior on Nomad. More specifically, we simulated the migration of a process every time a problem with some workstation is detected by a remote workstation that can offer help. To simulate the worst possible scenario, we assumed processes with the smallest possible images (288 KBytes), forcing the largest possible number of migrations when excessive memory demands is the problem. Every migration penalizes the source and destination workstations with 0.030 and 0.098 seconds, respectively, which are the times taken on these workstations to migrate a 288-KByte process in our system [12]. We let a period of overdemand stand for 5 seconds (same threshold as used in Nomad) before migrating a process away from a workstation. To further worsen the scenario, we assume that migrated processes run (and take up resources) forever at the destination workstations.

Table 3 shows the results of this experiment. From left to right, the columns of the table list the workstation name, the number of minutes when at least one resource was under excessive demand without Nomad, the number of minutes when at least one resource was under excessive demand with Nomad, the percentage reduction in resource overdemand, the number of processes migrated to the workstation, and the number of processes migrated away from the workstation. The "Total" row lists the sum of the overdemand periods without and with Nomad, the percentage of this time reduced by Nomad, and the total number of processes migrated in and out of workstations.

The results in the table clearly demonstrate that, even under a worst case scenario for Nomad, the system would have been able to significantly reduce the periods of resource overdemand. All workstations would have improved their resource utilization with Nomad, except for Brain which did not exhibit any problems without Nomad. Nevertheless, even in the case of Brain, the period of resource overdemand caused by Nomad amounted to no more than 2 minutes in 7 days. Overall, the cluster would have experienced a 99% reduction in the time at least one resource was exhausted by using Nomad. As a result of this improvement, applications should perform better under our system.

| Machine | Overdemand Without Nomad | Overdemand With Nomad | Reduction | Migs In | Migs Out |
|---|---|---|---|---|---|
| Atto | 9555.32 | 1.83 | 99.98% | 0 | 10 |
| Brain | 0.00 | 0.10 | —— | 61 | 0 |
| Gin | 941.56 | 11.54 | 98.77% | 21 | 30 |
| Kilo | 85.02 | 15.89 | 81.31% | 29 | 34 |
| Ripple | 780.52 | 2.72 | 99.65% | 24 | 32 |
| Rum | 1455.35 | 2.86 | 99.80% | 14 | 22 |
| Rye | 1532.04 | 21.13 | 98.62% | 28 | 37 |
| Scotch | 522.68 | 5.18 | 99.01% | 45 | 49 |
| Vodka | 1029.37 | 15.23 | 98.52% | 30 | 38 |
| Total | 15901.86 | 76.49 | 99.52% | 252 | 252 |

Table 3: Results of the simulation of Nomad.

Note however that a definitive evaluation of the actual performance improvements achievable by our system requires a complete implementation of it. Furthermore, we studied the behavior of Nomad assuming an academic cluster environment that may not be representative of other types of environments, such as heavy-duty scientific computing installations. A definitive evaluation of the system should take these other cluster environments into consideration as well.

# 4 Related Work

The migration strategy designed for Nomad is completely novel. However, several distributed operating systems (e.g. [6, 9, 8, 2]) share some of the same goals, policies, or mechanisms of Nomad. Here we concentrate on the operating systems that are most closely related to Nomad: GLUnix and Mosix. GLUnix seeks to execute interactive sequential and distributed applications efficiently on a cluster of uniprocessor workstations, keeping the Unix I/O semantics unchanged, providing static and dynamic load balancing, and detecting one failure at a time. GLUnix may have scalability problems since it uses a centralized server to assign unique process identifiers, to co-schedule distributed applications, to keep the state of all workstations in the cluster, to make decisions about process migration, and to detect failures. GLUnix is under development at UC Berkeley and, in its current version, does not achieve its goals completely.

Nomad differs from GLUnix in several ways. Nomad considers multiprocessor workstations, considers all aspects of a workstation's load, incorporates a distributed file system, works in a non-centralized fashion without the use of extra messages, and recovers from faults instead of just detecting them.

Mosix is also targeted at the efficient execution of sequential and distributed applications in clusters of uniprocessor workstations. Load balancing in Mosix is more sophisticated than in GLUnix, but only CPU and memory usage are considered. Migration decisions are based on load information messages periodically exchanged among a dynamic subset of workstations. Mosix can either migrate pages or whole processes. In case of memory problems, Mosix avoids swapping pages to disk by migrating processes to other workstations, but disregards the CPU utilization at the target workstations. Page migration is done directly to the target workstation's memory. However, when the CPU utilization is high on the source workstation, the whole process is migrated to the destination.

Nomad also differs from Mosix in several ways. Nomad considers multiprocessors, considers a larger set of cluster resources, incorporates a distributed file system, never migrates pages, considers the load on the target workstation before migrating a process to it, and co-schedules concurrent applications. In terms of their migration strategies, Nomad uses the file access patterns to guide the dissemination of load information, while not involving any extra messages to implement the actual load information exchanges. Although for many cluster configurations it is unlikely that the extra messages in Mosix should cause serious overheads, our work shows that these extra messages are unnecessary given a distributed file system.

# 5 Conclusion and Future Work

This paper presented a short introduction to Nomad, an efficient operating system for clusters of uni and/or multiprocessors designed and implemented in the context of an MSc thesis at the Federal University of Rio de Janeiro. The main goal of Nomad is to efficiently support (high-performance or interactive) parallel, distributed, and sequential applications. Nomad includes several important characteristics for modern cluster-oriented operating systems, including scalability, efficient resource management across the cluster, efficient scheduling of parallel and distributed applications, distributed I/O, and fault detection and recovery. Nomad does not involve any extra messages for resource management, distributed scheduling, and fault tolerance, taking advantage of the I/O traffic associated with its distributed file system.

A preliminary evaluation of the load balancing aspect of Nomad showed that the pattern of file accesses produced by Nomad's distributed file system and real workloads can effectively be used as a mechanism for distributing load information across the cluster. In addition, our results show that Nomad can almost eliminate the periods of excessive demand for resources by intelligently migrating processes.

Based on these results, we expect the complete implementation of Nomad to be efficient and to become an interesting foundation for research on distributed operating systems for clusters of workstations. Our future work includes completing the implementation and evaluation of the system. Right after this, the Nomad source code will be made available to the public for non-commercial use.

# References


[1] Andrea C. Arpaci-Dussau, David E. Culler, and Alan M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Madison, Wisconsin, June 1998.

[2] Amnon Barak and Oren La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, Mar 1998.

[3] Eliseu M. Chaves and Valmir C. Barbosa. Time sharing in Hypercube Multiprocessors. In *Proceedings of 4th IEEE Symposium on Parallel and Distributed Processing*, pages 354–359, Arlington, TX, Dec 1992.

[4] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, Mar 1988.

[5] Toni Cortes. Software RAID and Parallel File Systems. In Rajkumar Buyya, editor, *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, 1999.

[6] Fred Douglis and J. Ousterhout. Transparent Process Migration: Design and Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(8):757–785, Aug 1991.

[7] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2), 1998.

[8] D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. *Software: Practice and Experience*, Feb 1998.

[9] Yousef A. Kalidi, José M. Barnabéu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris MC: A Multi Computer OS. In *Proceedings of 1996 USENIX Conference*, January 1996.

[10] E. L. Miller and R. H. Katz. RAMA: An Easy-to-Use, High Performance Parallel File System. *Parallel Computing*, 4(23):419–446, June 1997.

[11] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[12] Eduardo Pinheiro. Nomad: An Efficient Operating System for Clusters of Uni and Multiprocessors. Master's thesis, COPPE Systems Engineering, Federal University of Rio de Janeiro, August 1999. In Portuguese.

[13] Sun Systems. *Sunos and Solaris Reference Manuals*. Sun Systems, Inc.